Visual Exploration of Function Call Graphs for Feature Location in Complex Software Systems

Johannes Bohnet
University of Potsdam
Hasso-Plattner-Institute
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
bohnet@hpi.uni-potsdam.de

Jürgen Döllner
University of Potsdam
Hasso-Plattner-Institute
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
doellner@hpi.uni-potsdam.de

Abstract

Maintenance, reengineering, and refactoring processes of software systems are typically driven and organized in terms of features. Feature change requests need to be translated into changes in source code, which is a highly cost intensive and time consuming task when complex legacy software systems are concerned; their documentation is likely to be outdated and incomplete. In this paper, we propose a prototype tool that supports users in locating and understanding feature implementation in large (>1 MLOC) C/C++ systems. A combination of static and dynamic analysis allows extracting of the function call graph during feature execution and interpreting it within the static architecture of the system. An interactive multi-view visualization enables users to explore that graph. An effective 2½D visualization provides various visual cues that facilitate finding those paths in the function call graph that are essential for understanding feature functionality. Additionally to source code aspects, the dynamic metric of function execution times is exploited, which gives significant hints to feature-implementing functions. Furthermore, information on functions is extended by architectural aspects, thereby supporting users in remaining oriented during their analysis and exploration task as they can give priority to selected architectural components and thereby hide insignificant function calls.

CR Categories: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering*, H.5.2 [Information Interfaces and Presentation]: User Interfaces – *Graphical user interfaces (GUI)*

Keywords: Dynamic Analysis, Feature Location, Feature Analysis, Program Comprehension, Reverse Engineering, Software Visualization

1 Introduction

In 1994, Waters and Chikofsky stated that “year after year the lion's share of effort goes into modifying and extending preexisting systems, about which we know very little” [Waters and Chikofsky 1994]. Today, 12 years later, this striking statement increasingly proves to be true for IT systems engineering. Requests for modifying and extending software systems are typically motivated and expressed by end-users in terms of features, i.e., an observable behavior that is triggered by user interaction. To fulfill typical change requests developers need to translate them into terms of source code. That is, they firstly have to locate the source code components that are responsible for feature functionality, and secondly they need to understand how these components interact. Where legacy software systems are concerned, this is a cost intensive and time consuming task as legacy systems are typically complex systems. The term complex refers (a) to the system implementation consisting of several millions of lines of code (LOC) and (b) to the large number of changing developers who design and maintain the system over decades. During the long evolution period the system’s documentation from design time is likely not updated continuously. Therefore, the system’s implementation often remains as the only reliable documentation.

Slicing techniques [Weiser 1981; Xu et al. 2005] provide ways to reduce the amount of source code a developer has to inspect to locate and understand feature implementation. Static slicing techniques reduce the code to statements that may affect the calculation at a specific code location or that are affected by a calculation at a specific code location (backward or forward slice, respectively). Dynamic slicing techniques further reduce the source code by only allowing for a given set of input variables or given user inputs, respectively. That is, they take only one specific execution path into account.

Where object-oriented software systems are concerned, some dependencies between code components are difficult to reveal when only analyzing the source code [Chen and Rajich 2001]. By polymorphism and dynamic binding, dependencies may not be defined until runtime. Furthermore, event driven systems, often used for graphical user interfaces (GUI) or network communications, are based on dynamic callback mechanisms. Logging function calls during runtime overcomes these limits. It enables users to understand which functions were executed and how they were interacting. In the context of analyzing feature implementation this technique permits location of feature-relevant code components. Additionally, it provides information for an initial understanding on how feature functionality is implemented. However, a deep understanding of the implementation requires further techniques, e.g., debugging techniques, as function call logging neglects interaction of code components based on shared data structures.

The exploration of a dynamically derived function call graph is typically performed by means of a step-by-step analysis from one function to another. The user’s aim is to identify a path to those functions that implement the relevant functionality. Each step requires an assessment about the function call’s importance for understanding feature functionality. Existing analysis tools

SOFTVIS 2006, Brighton, United Kingdom, September 04–05, 2006.
© 2006 ACM 1-59593-464-2/06/0009 $5.00

commonly support users during their exploration task by means of providing fast access to the textual source code representation of the function calls. By analyzing the code, a user decides whether a function call is essential for understanding the analyzed functionality or not. This decision is based on criteria such as: (a) the called function's name; (b) the function calls that follow this call; (c) calculations that are performed between the calls, including accesses to global or member variables.

Additionally to these source code based criteria, runtime data based criteria can help the user in finding a path to the essential functions that implement the feature's core functionality. One such runtime criterion is the information on the execution time of a function. Functions with long execution times most likely induce the execution of other functions that implement calculation intensive functionality, i.e., typically the feature's core functionality. Whereas functions with short execution times are likely to be just helper functions or functions that collect data from member variables. Therefore, function execution time represents a reasonable metric for assessing the function call's importance for feature understanding in an anticipatory way.

A further criterion for assessing a function's importance is knowledge of the static architecture of the system. A user can refrain from further analyzing a function call if the call leads to an architectural component that has earlier been classified as dispensable for feature understanding. Such an architectural component might be a component encapsulating IO functionality while the user is trying to understand the feature of coloring text in a word processor application.

In this paper we present a prototype tool for analyzing feature implementation of large (>1 MLOC) C/C++ software systems. A combination of dynamic and static analyzing techniques permits extraction of the function call graph during feature execution and combines it with information on the static architecture of the system. The user explores the function call graph within an interactive visualization system that consists of various textual and graphical views. During the analysis of the function call graph, the user receives support with deciding which function call to follow, so that she/he finally gets to feature relevant functions. This support is achieved by various concepts:

a) Cutting a subgraph out of the function call graph. This process is based on the functions' execution times as a metric for assessing the function's importance for locating the core implementation of the feature;
b) Combining the function call graph with information on the static system architecture;
c) Providing fast access to source code and synchronizing both views: the view on the function call graph and the view on the source code;
d) Support for bookmarks of already explored functions.

Prerequisites for analyzing a software system with our tool are (1) the availability of the source code written in C/C++, and (2) an executable file that is compiled with debug information and can be executed under a GNU/Linux operating system (OS). No additional source code instrumentation is necessary. The tool has been tested with software systems consisting of over a million LOC.

The remainder of the paper is organized as follows: Section 2 discusses related work. Section 3 describes the process of analyzing feature implementation with our prototype tool and explains how data is extracted from the software system. The next Section illustrates how the tool supports the user during function call graph exploration. It describes how criteria that influence the user's exploration steps are mapped into visual cues within the multi-view visualization system. Section 5 finally concludes this paper.

2 Related Work

Software Reconnaissance [Wilde et al. 1992; Wilde and Scully 1995] is a technique for locating feature implementation based on a comparison of traces from test cases with and without feature execution. Prerequisite for applying this technique is the instrumentation of the system's source code. Lukoit et al. built the visualization front-end *TraceGraph* [Lukoit et al. 2000] for the *Software Reconnaissance* analysis process. It displays the activity of source files, subroutines, or statements color encoded per time interval. In contrast to our approach no relationship between these components is shown. Other research on locating feature implementation is based on *Program Dependency Graphs* [Ottenstein and Ottenstein 1984], which are extracted by statically analyzing source code. *RIPPLES* [Chen and Rajich 2001] supports the user during manual exploration of a dependency graph by 2D graph visualization. The user decides whether a component, i.e., function, basic block, or statement, is relevant for the feature and adds it to the *search graph* that finally represents the feature implementation. Contrary to our approach, neither hints for assessing the function's importance for understanding feature functionality is given, nor the functions are related to architectural high-level structures. Eisenbarth et al. [2003] first compare dynamic execution traces, which depend on a set of features, by applying *concept analysis* to find out to which feature a computational unit contributes. Later the user identifies additional feature specific units by exploring the statically extracted dependency graph. Our approach, which suggests the interpretation of functions within the system architecture, is orthogonal to the described techniques above and can, accordingly, be combined with them. Furthermore, our user-supporting visualization technique may be used as visualization front-end.

A variety of visualization techniques are used to facilitate reverse engineering tasks. The *SeeSoft* technique [Eick et al. 1992] displays LOC metrics as color encoded miniaturized source code lines. Various other techniques are based on this idea, e.g., *Tarantula* [Jones et al. 2002], *Gammatella* [Orso et al. 2003], *Bee/Hive* [Reiss 2001]. *sv3D* [Marcus et al. 2003] extends the technique to three dimensions. Other techniques, such as ours, use graph visualization techniques. The *Rigi* [Müller et al. 1993] reverse engineering environment displays the hierarchical software architecture as a 2D layout of nested boxes with connecting straight lines. *SHriMP* [Michaud et al. 2001] enhances Rigi views by means of elaborate navigation and exploration techniques and is used for visualizing statically analyzed Java programs. *CodeCrawler* [Demeyer et al. 1999] calculates various software metrics and encodes them in box shapes of simple graphs. Several graph visualizing techniques use virtual reality techniques: *NV3D* [Parker et al. 1998] displays nested cubes connected by tubes and is used to visualize execution traces with animated arrows moving along the tubes. *CrocoCosmos* [Lewerentz and Simon 2002] creates a universe-like visualization of software components by a force-directed layout technique

based on static software metrics. Similarly, *JST* [Irwin and Churcher 2003] creates a software component universe by parsing Java source code and applying metrics. In the case of both approaches, the user explores the data by applying standard navigation techniques of a VRML browser. As our approach proposes, some visualization techniques use the landscape metaphor for presenting software systems structure and behavior. Zhou et al. [2003] visualizes message flows in massively parallel supercomputers. Boxes regularly positioned on a plane represent processors and are connected by arcs. Balzer et al. [2004] map the architecture of Java programs as nested box and sphere shapes positioned on a plane. Relations between components are represented by connecting arcs. This differs from our approach in that no self-organizing layout for encoding information on component relations is used.

3 Analysis Process

The prototype tool we present enables the user to efficiently locate those parts of the source code that correspond to the implementation of the analyzed/explored feature. Thereby developers get support with building up a mental map of the high-level concepts implicitly contained in the system's coding.

The analysis process of features is divided into three steps as illustrated in Figure 1:

1) In an automated step the static structure of the source code is extracted according to its directory and file structure. This initial model can be manually refined by the user.

2) The user (1) chooses a scenario, i.e., a sequence of user-system interactions that triggers the feature execution, and (2) applies the function-call logging mechanism while executing the scenario. Prerequisite for performing this step is only the availability of the system's executable file compiled with debug information. No code additional instrumentation is necessary.

   Next, the function call graph is automatically reconstructed from the log file. Finally, the functions are located within the system architecture.

3) An interactive multi-view system visualizes the function call graph. A combination of textual and graphical views enables the user to efficiently explore the graph. The user receives support in her/his decision which path in the graph to follow. This is achieved by (a) a technique that cuts out a subgraph from the function call graph based on the function currently in the focus of analysis and on the functions' execution times, by (b) a graph layout that reflects the functions' dependencies and additionally provides architectural information, and by (c) a fast and synchronized access to source code representation.

For reasons of clarity we illustrate the analysis process of our prototype tool by means of a feature analysis of the *Firefox* browser application from the *Mozilla* open source project [Mozilla], which is written in C/C++ and consists of approximately 2.600.000 LOC. The feature we analyze is requesting a web page for a specific URL.

The following sections describe in detail the data extraction processes from the software system. Section 3.1 explains how the function call graph and profiling information on the function execution times are gathered. Whereas section 3.2 describes how

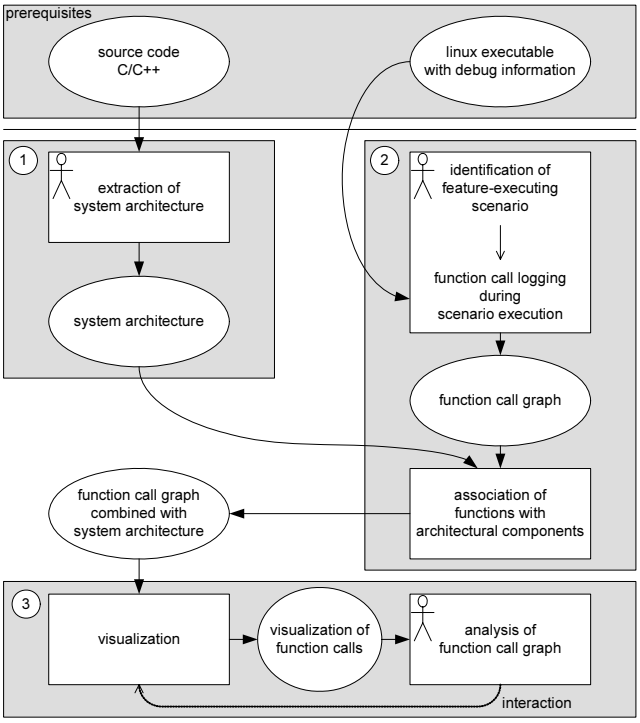

Figure 1: Overview of the semi-automated analysis process of feature analysis. The user-symbol indicates activities with user interaction.

an architectural model of the system is derived from the source code.

3.1 Call Graph Extraction

Logging function calls of a software system during runtime permits recapitulation of the internal system behavior once a specific feature has been executed. This dynamic technique overcomes some of the limitations of static source code analysis techniques as it gives a holistic picture of the analyzed system within the runtime environment. Function calls to and from shared libraries are detected, which is important when analyzing features that are triggered by event driven GUIs. What is more, this technique solves typical comprehension problems during static analysis due to dynamic binding and polymorphism. Additionally, on the basis of dynamic analysis the user has information on the time spent for executing a function. This metric allows assessing in an anticipatory way, whether a function induces the execution of further calculation intensive functions, which may implement the feature's core functionality. In the opposite case, the function call returns rapidly after initiating the execution of just a few other functions.

Contrary to most static approaches, function-call logging analyzes only one specific execution path and is therefore incomplete. As a further disadvantage this dynamic technique is not able to reveal dependencies between two functions that origin in the access to the same data structure. Moreover, the logging mechanism affects the system's behavior as additional instructions need to be added to the system to produce log entries. This code instrumentation is either done before compile time by source code instrumentation or after building the executable file by binary code instrumentation [Nethercote 2004]. Where Java applications are concerned, a

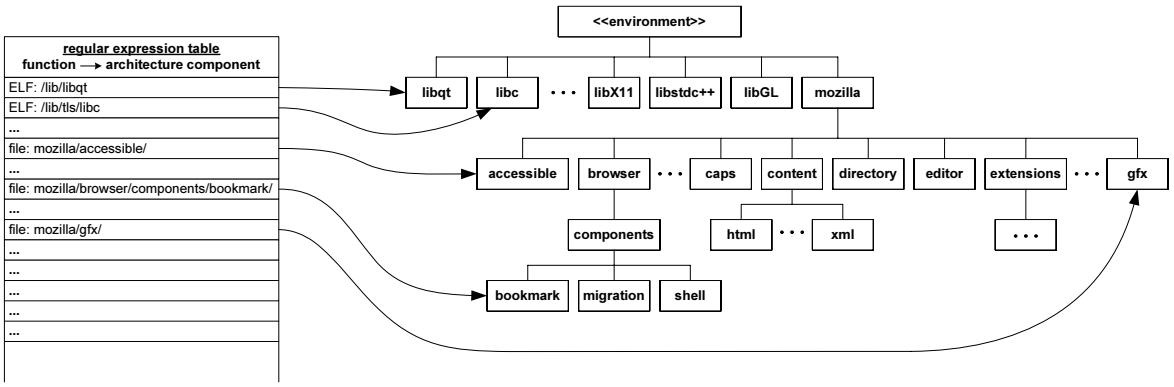

Figure 2: Architecture model of Mozilla Firefox browser extended by components representing shared libraries of the OS. The regular expression table defines how to associate functions with model elements. Expressions relate to the functions' debug information.

common mechanism for logging and profiling is to address the Java Platform Debugger Architecture [JPDA]. This way, no code instrumentation is necessary. Where C/C++ applications executed on a Linux-OS are concerned, a similar logging mechanism without source code instrumentation can be implemented by starting the application within a wrapper process. By just-in-time disassembling and reassembling of the application, the wrapper process is able to analyze the application's behavior. However, all dynamic analysis techniques slow down the analyzed system. Depending on the amount of logged information, the analyzed system can register a change in behavior. If the user analyzes realtime behavior, the loss in performance is crucial. However, other behavior can also be affected. At worst, the system will no longer react to user interaction properly.

Our prototype tool uses Callgrind [Weidendorfer et al. 2004] for function call logging during runtime. Callgrind is a profiling tool based on Valgrind [Nethercote and Seward 2003], which is a suite of tools for debugging and profiling Linux applications. As described above, Valgrind works as a wrapper process for the application. Hence, no source code instrumentation is necessary. Callgrind is a light-weight profiling tool that does not log the full sequence of function calls but summarizes call relations and reconstructs the function call graph from the summaries. The logging mechanism is enabled and disabled interactively by the user, permitting only those function calls to be logged that are executed during feature execution.

The process of extracting the function call graph is illustrated in Figure 1 (step 2). First, the user identifies a scenario that triggers feature execution, i.e., a sequence of user interactions is identified. Second, the user performs these interactions while the logging mechanism is enabled. After this manually executed part of the analysis process, the function call graph is automatically reconstructed from the log file. Additional information on the function execution times is as well extracted from the log file.

For our exemplary analysis of the Firefox browser, this means that we start the browser, type in the desired URL, then we enable the logging mechanism, press the return key, wait until the web page is displayed, and finally disable the logging mechanism. Even such a short logging time results in a function call graph of more than 20.000 interacting functions and several million call relations.

Before the function call graph is passed to the multi-view visualization system, the graph is combined with information on the static architecture of the system. This process is explained in detail in the next section.

3.2 Extraction of the System Architecture

For the system to be explored we have to extract the model of the system's static architecture from its source code. With forward engineering, these models typically consist of classes, packages, and relations among them.

- By *low-level models* we refer to models up to class level abstraction; a variety of tools exist [Borland Together; IBM Rational Software] for an automated reconstruction.
- By *high-level models* we refer to mechanisms of structuring code units (e.g., C++/Java: namespaces/packages) provided that the system makes use of these mechanisms.

Other methods of structuring classes to high-level code units that do not depend on syntax of the programming language are (a) the use of naming conventions or (b) the organization of source files in systematic directory structures to reflect system partitioning. Structuring by source files is often used in large software systems, and we exploit it in our approach for architecture reconstruction. An initial model is created from the directory structure automatically. As the user typically has additional architectural knowledge of the system, the model can be refined manually in a second step.

In a final step the architecture model of the system is extended to a holistic model that also reflects shared libraries (precisely: ELF objects, the Executable and Linking Format) of the runtime environment. This way, all functions of the dynamically extracted call graph can be associated with architectural components and the user is enabled to understand function interaction from an architectural point of view. Specific low-level function interaction might be interpreted as interaction, for instance, between a specific architectural component of the system and the GUI library of the operating system.

To associate functions from the call graph with architectural components of the architecture model, pattern matching is used. In a similar way as proposed by Murphy et al. [1995], the user builds up a table of regular expressions that relate to the debugging information on the function:

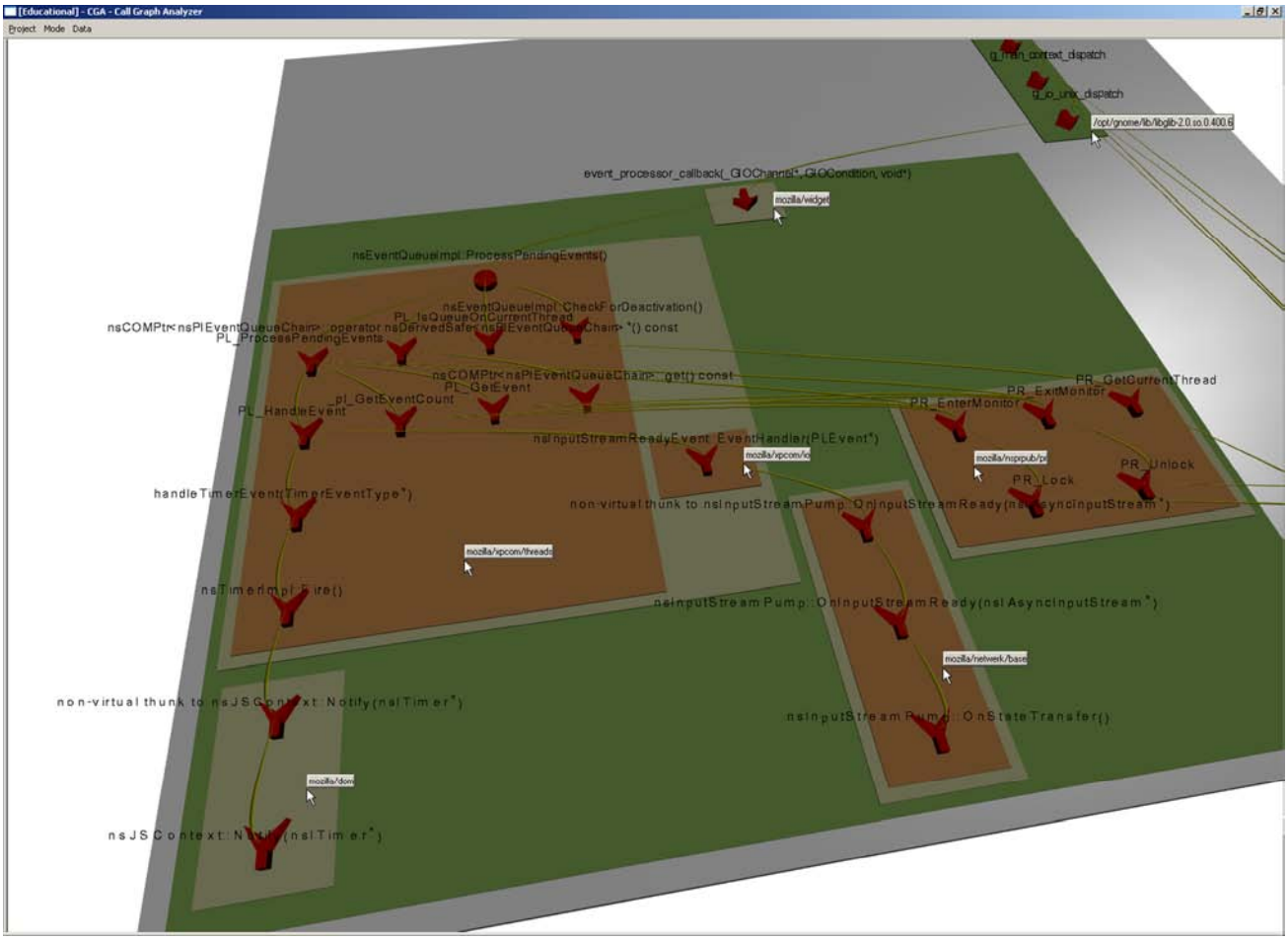

Figure 3: Graph Exploration View: The function currently in the focus of analysis and its neighborhood are displayed within the static system architecture. The focus function is represented as a disc, predecessor functions as arrow tops and successor functions as arrow bottoms. A function call is represented as an asymmetrical arc. Caller and callee can be distinguished as the arc's peak point is shifted to the destination of the call.

1) The function's full name, including namespace and class prefixes;
2) The file where the function is implemented;
3) The ELF object into which the function has been compiled.

Typically the number of entries of such a table of regular expressions is comparable to the number of architectural components. Advantages of using regular expression for defining the interface between architecture model and function call graph are (a) the simple and fast creation and (b) the flexibility with that the user can associate functions to architectural components. Figure 2 illustrates the architecture model and the regular expression table for the example of analyzing the Firefox browser.

4 Visual Call Graph Exploration

Before users can modify or enhance existing features in software systems, they need to locate those code components that implement the feature. Extracting the function call graph from the running system while the feature is executed, presents a first step of the locating process as it reduces the search space the user has to inspect. However, the size of such a function call graph may still be immense. For complex +100k LOC software systems the graph typically consists of tens of thousands interacting functions. Therefore, the graph needs to be further reduced to the feature-relevant functions. The visualization front-end of our prototype tool provides various cues for identifying feature-relevant functions thereby facilitating the reduction process for the user. The user explores the function call graph within an interactive multi-view system. Various synchronized textual and graphical views support the user during the analysis process. In addition to views that give information on the call graph and the static system architecture, the tool provides a view for fast access to source code information, necessary for deciding whether a function is important for feature understanding or not.

In the following sections the different views of our prototype tool are discussed.

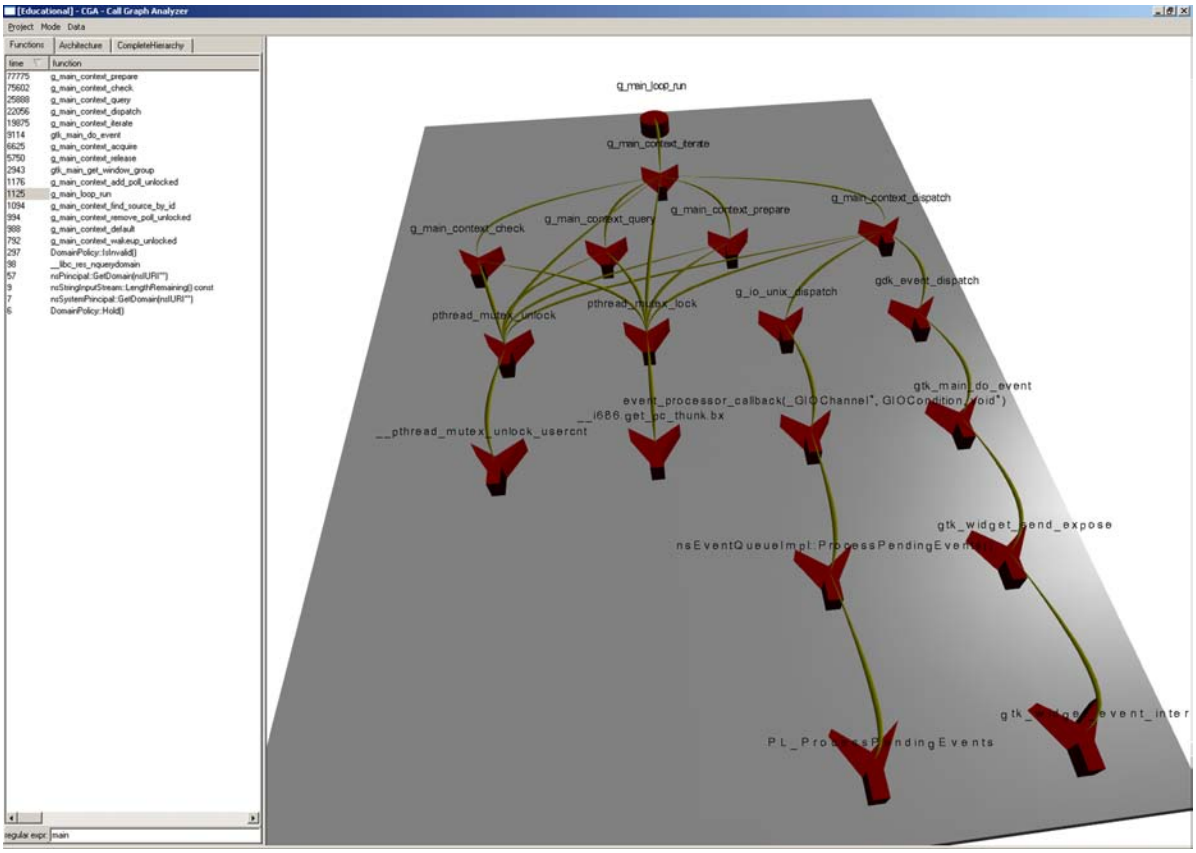

Figure 5: Neighborhood of function `g_main_loop_run`. The layout does not reflect the system architecture.

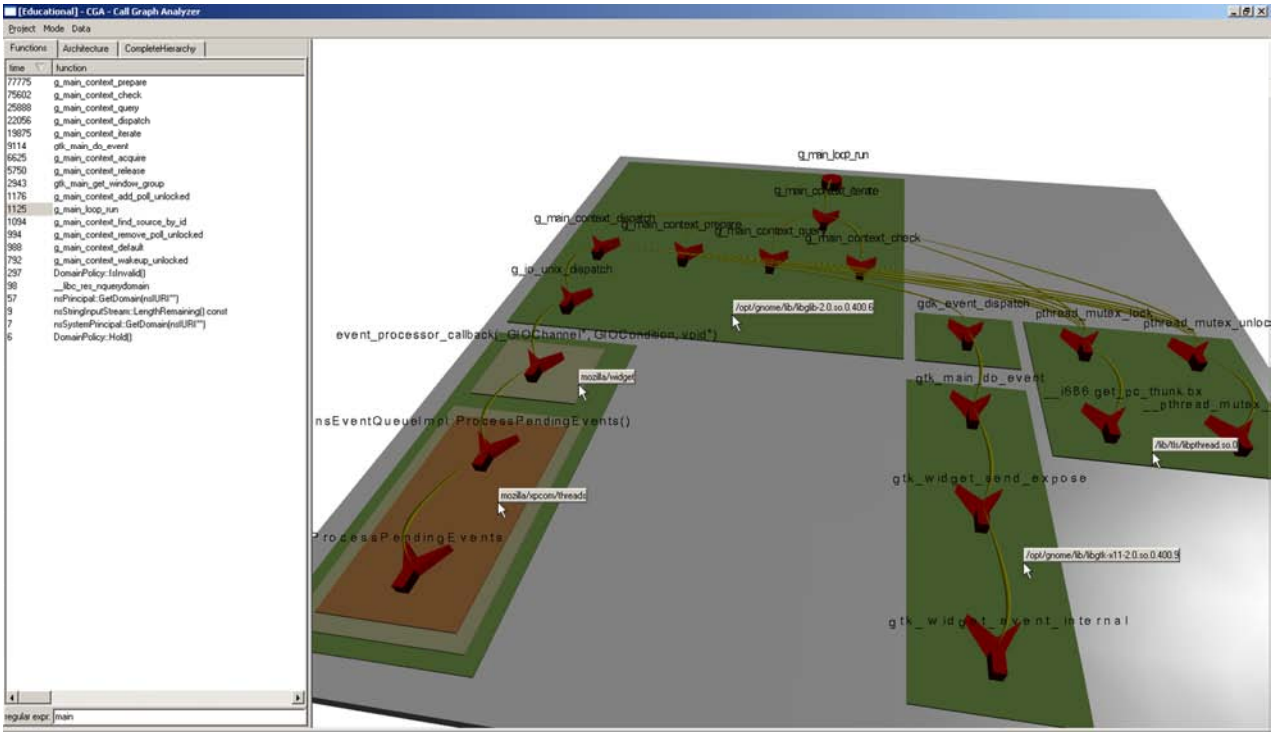

Figure 4: Neighborhood of function `g_main_loop_run`. The layout reflects the system architecture. Therefore, users can, firstly, identify the function call chain that leads to feature implementing code and, secondly, understand how the control flow enters the mozilla system through the *mozilla/widget*- and *mozilla/xpcom/threads*-components.

4.1 Graph Exploration View

The graph exploration view is the key view during the exploration task of the function call graph. It provides various kinds of information on the function that is currently in the focus of the user's analysis: most importantly the function's "neighborhood" and the *architectural context* of the functions, i.e., the components of the static system architecture to which the functions belong.

A *function's neighborhood* is a set of functions that directly or indirectly pass the control flow to the function (predecessor functions) and of those that receive the control flow from it (successor functions). Typically, the function call graph consists of a huge amount of connected functions. Even the subgraph that represents the neighborhood of a specific function can still be large. The set of all functions that are connected to a specific function by at most 5 calls, for instance, usually consists of more than several hundred functions. Understanding the behavior of the system from such a complete function neighborhood is a complex task, where the user needs to decide which the important functions for feature understanding are.

In our tool the complete function neighborhood is further reduced to a set of functions that are likely to be of high interest for the user. This assessment is based on the execution time spend for a function call. This metric reflects a call's importance, because a long execution time means that the call is likely triggering further calls that lead the feature's core functionality. Whereas a short execution time indicates that only a few further function calls succeed this call and the called function is likely to be just a helper function or it only collects data from member variables. The reduced neighborhood (in the remainder shortly termed as "neighborhood") consists of all direct predecessor and successor functions. From all indirectly connected functions only those with a relatively large execution time belong to the neighborhood. To define the second generation successors, for instance, all outgoing calls of a first generation successor (directly called by the function) are analyzed. Only the n ($n \in \mathrm{N}$) functions with the largest execution times of all n calls belong to the neighborhood. The Parameter n decreases with the order of successor or predecessor generations, respectively. It is set zero for all generations that are connected to the central function by more than g_{max} ($g_{\max} \in \mathrm{N}$) indirections. Therefore, the neighborhood is a graph of connected functions where every function can be reached from the central function by at most g_{max} function calls.

4.1.1 Visualization

The function in the focus of interest and the functions of its neighborhood are displayed as shapes placed on a plane. The focus function is represented as a disc, predecessor functions as arrow tops and successor functions as arrow bottoms. Calls between functions are represented as connecting arcs with an asymmetrical shape. The arc's peak is shifted to the destination function of the call, so that the user can easily distinguish between caller and callee. Figure 3 illustrates the usage of the different shapes.

To enable the user to relate the shapes with their belonging function, the shapes are equipped with a text label displaying the function's name. The labels are 3D shapes, always oriented to the viewer. This 2½ dimensional technique permits the usage of arbitrary shaped geometries that need to be displayed with textual information. More information than that displayed by the labels can interactively be gathered with the mouse (see section 4.1.2).

As explained in section 3.2, the function call graph is related to the static system architecture. We exploit this relationship for the visualization, in that functions are grouped accordingly to the architecture component that they are associated with. Moreover, architecture components are explicitly visualized as rectangle shapes to clarify the boundaries of such a group of functions. These rectangles are displayed in a nested way to reflect the hierarchical structure of system architecture. By this, the user can easily see whether two functions, that are associated with different architecture components, belong to the same parent component. The shapes' layout is based on layout techniques provided by the GraphViz-library [Ellson et al. 2001].

To illustrate the advantage of visualizing both the function call graph and its architectural context, Figure 5 shows the neighborhood of function `g_main_loop_run` without providing the architectural context. Whereas Figure 4 shows the same set of functions with a layout that provides information on the system architecture. In the figure without the architectural context the user needs to read most of the functions' names to be able to decide which function should be taken next into the focus of the analysis. Whereas in Figure 4 the user receives effective support with identifying those function calls that are important to locate and understand feature implementation. The function chain that leads to feature relevant code is:

```
g_main_loop_run → …
→ g_io_unix_dispatch
→ event_processor_callback()
→ nsEventQueueImpl::ProcessPendingEvents()
→ PL_ProcessPendingEvents
```

Furthermore, the user is able to build up a mental map of the high-level concepts implicitly contained in the system's coding: The control flow origins in the *glib*-library and enters the *mozilla* system through the *mozilla/widget*-component, which passes the flow to the *mozilla/xpcom/threads*-component.

For reasons of clarity, we continue to illustrate the user's analysis process by providing figures that demonstrate how the user explores the function call graph. After identifying the function call chain that enters the mozilla system, the user takes function `nsEventQueueImpl::ProcessPendingEvents()` into the focus of the analysis by double-clicking it with the mouse. Figure 3 shows the resulting view. The call chain that origins in the *glib*-library and enters the *mozilla* system is still visualized. However, now the successor functions of the focus function are displayed. Suppose the user is interested in understanding how *mozilla* implements network functionality. This view shows her/him, firstly, that functions from the *mozilla/network*-component are active and, secondly, that the control flow is passed from the *mozilla/xpcom/threads*- via the *mozilla/xpcom/io*- to the *mozilla/network/base*-component. Next, the user double-clicks the `nsInputStreamPump::OnInputStreamReady()` function. Then function `nsInputStreamPump::OnStateStart()` is chosen. Figure 6 shows the result. It is easy to identify how the control flow passes from the *mozilla/network/base*- via the *mozilla/network/protocol/http*- to the *mozilla/uriloader/base*-component.

4.1.2 Interaction

Users may interact with the visualization system for 3 purposes:

1) Gathering additional information on a function, an architecture component, or a function call;
2) Navigating within the displayed space;
3) Taking a different function into the focus of analysis.

Additional information on a function is gathered interactively by pointing with the mouse on a function, an architecture component, or a function call shape. A small window pops up containing detailed information. In the case of a function shape it displays the function's full name including namespace and class prefixes. For an architecture component shape it displays the component's name and the parent components' names. Pointing onto a call relation shows (a) how often the call was performed and (b) its execution time.

Further mouse interaction is used for navigating within the displayed space. Typically, the user is firstly interested in a global perspective where all functions are shown. This allows quickly gathering an overview on which the active architecture components are and how they are interacting. However, to analyze the call relations of a specific function the user usually needs to zoom in and to move within the information space. Therefore, the visualization system allows navigating with three elaborate navigation techniques:

- Moving visual focus: Clicking on a function shape with the mouse pointer moves this function shape into the center of the display;
- Zooming: Using the mouse wheel zooms the perspective in or out, respectively. In combination with the first navigation technique, users can efficiently choose the level of detail and switch between focus and context;
- Tilting view angle: The user looks from a 3D perspective onto the function shapes placed on a 2D geometry. By tilting the view angle, the user can continuously vary the density of the displayed items. Function shapes in the front are displayed in full detail whereas function shapes in the back are displayed with less detail, however, still showing structural information such as to which architectural component they belong.

These navigation techniques help the user to efficiently gather information on function calls that are necessary to decide which function to analyze next. To step further in the function call graph, the user clicks two times on the function's shape she/he wishes to analyze next. Thereupon, this function becomes the center of the 2½D visualization and the layout of its predecessor and successor functions is calculated. Alternatively, the user can click on textual entries seen in the other views of the multi-view system.

The next sections describe the textual views of the interactive visualization system.

4.2 Source Code View

The graph exploration view provides various visual cues for facilitating the task of assessing a function's importance for

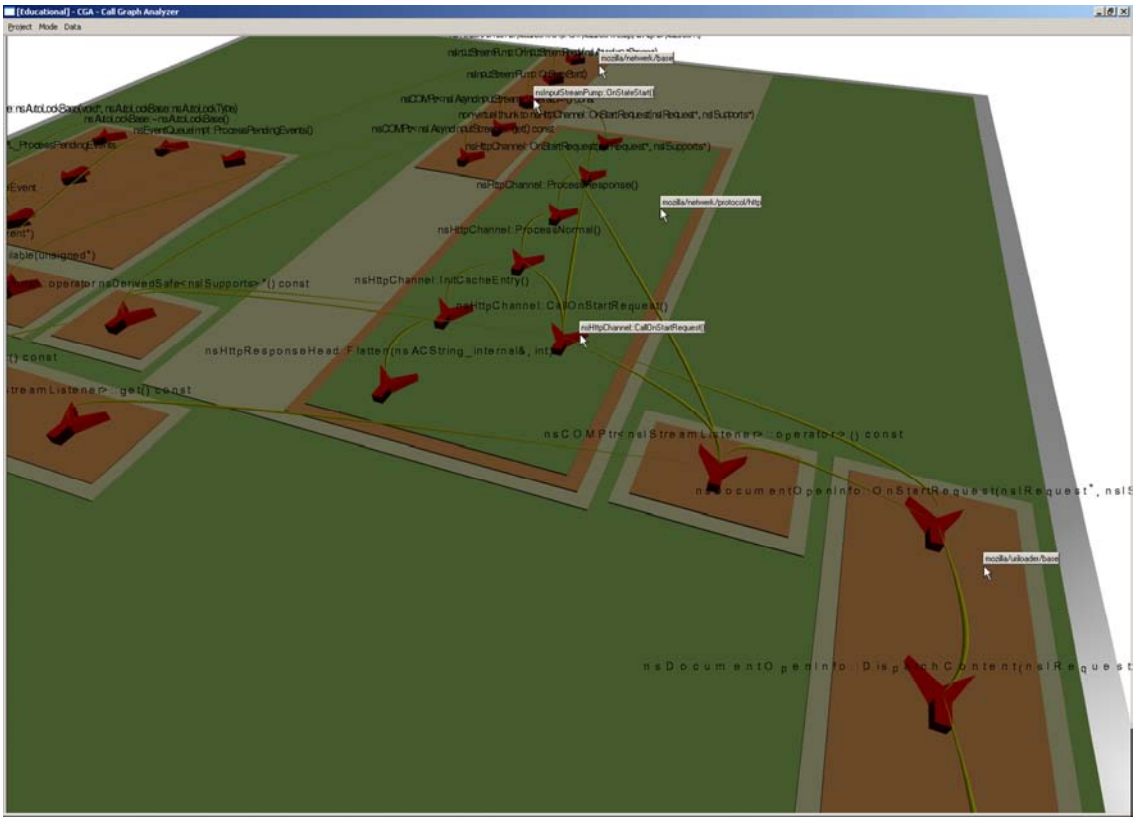

Figure 6:Neighborhood of function `nsInputStreamPump::OnStateStart()`. The control flow is passed from the *mozilla/network/base*- via the *mozilla/network/protocol/http*- to the *mozilla/uriloader/base*-component.

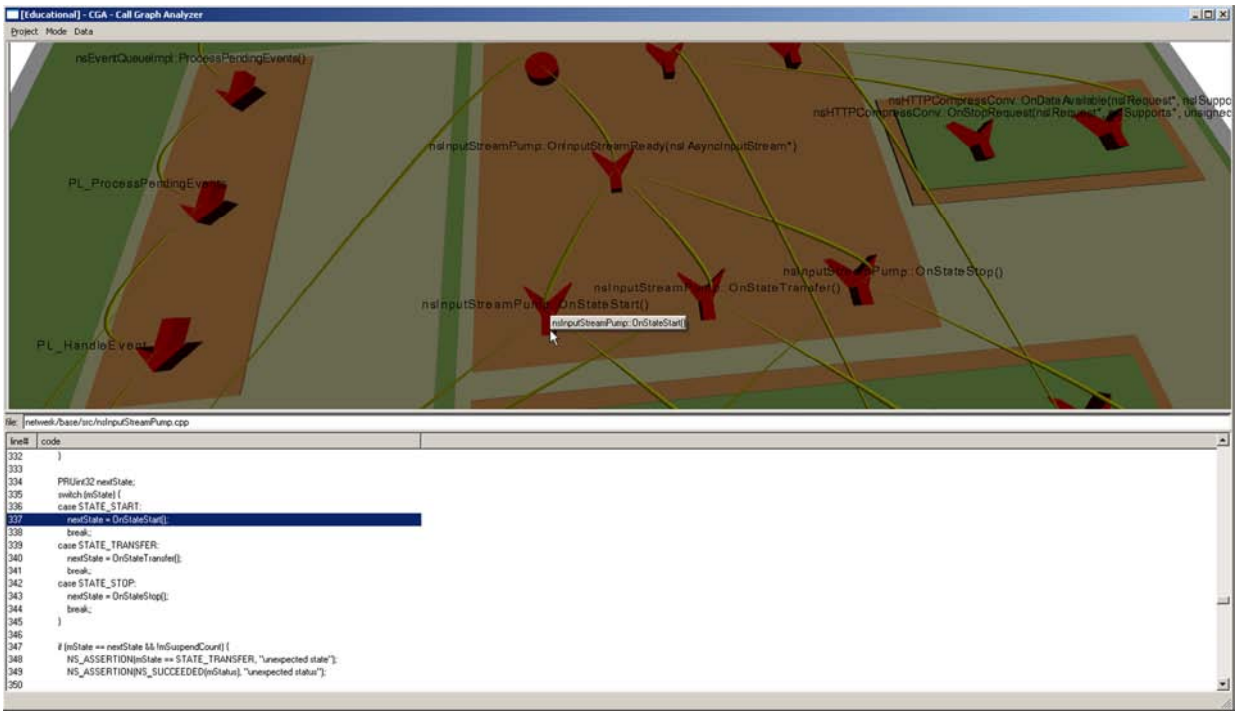

Figure 7: The textual *source code view* is synchronized with the graphical *graph exploration view*. If the user clicks with the mouse on a function shape or call relation, the corresponding source code file is loaded into the *source code view*, and the call's code line is highlighted.

feature understanding. However, this view can only guide the user and indicate those functions that should be analyzed first. For a profound decision whether to follow a specific path in the call graph or not, the user needs to analyze the source code. The source code view provides fast access to the code of the currently analyzed function. By selecting a function shape in the graph exploration view, the source code view is updated and it shows the code line with the call to the selected function. Vice versa, when the user selects a source code line that contains one or more function calls, the belonging shapes in the graph exploration view are highlighted. This synchronization of the two views enables the user to easily switch between the two representations: First, the user analyzes the function calls in the graph exploration view. If necessary, she/he then switches to the source code view that instantly provides information on the analyzed function call on code level representation. The view is illustrated in Figure 7.

4.3 Function Search View

The function search view contains all functions that were active during feature execution. These functions are stored in a list, which can be sorted by various criteria (execution time, name, architecture component, file name, ELF name). Furthermore, the user can search functions by pattern matching with regular expressions. This view serves for finding a suitable entry point into the function call graph. A reasonable starting point for feature analysis is typically a function with long execution time. This view is shown in Figure 4.

4.4 Bookmark View

The bookmark view supports the user in maintaining oriented during call graph exploration. Bookmarks can be set to mark specific functions that the user classifies as important or that should be analyzed in detail after performing other analysis tasks.

5 Conclusions

Features represent core elements in maintenance processes when large and complex software systems are concerned. Therefore, developers need to understand, which code components implement a specific feature, and how these components interact. Extracting the function call graph of a software system during feature execution at runtime, is of great assistance to developers for localizing and understanding feature implementation as it reduces the amount of code developers have to inspect. However, for identifying those functions that implement the feature's core functionality, developers need to step through the function call graph, each time deciding whether a function mainly contributes to the feature or not. Our approach facilitates this decision-making in two ways:

1) The prototype tool provides a condensed view on source code and runtime criteria necessary for assessing a function's importance for feature understanding. Particularly, the interpretation of the function execution time supports the developers to efficiently locate feature implementation.
2) Information on function calls is interpreted within the system architecture, which enables developers to refrain from analyzing a function call if it leads to an architectural component that has been identified as dispensable for feature understanding.

Thus, extracting function call graphs and then visualizing it in an architectural context appears to be an effective approach to gaining a better understanding of essential aspects of complex software systems and to speeding up related reverse engineering

tasks. Our approach could also be applied on software systems written in different programming languages than C/C++ such as Java. However, we focus on C/C++ legacy systems as we consider these systems the most critical ones concerning maintaining and reengineering tasks.

In further studies we plan to perform user evaluation of the prototype tool to identify potential improvements. Furthermore, we plan to consider the ideas proposed in [Eisenbarth et al. 2003; Wilde and Scully 1995] such as analyzing multiple execution traces to facilitate identifying those code components that dedicate mainly to only one specific feature.

References

BALZER, M., NOACK, A., DEUSSEN, O., and LEWERENTZ, C. 2004. Software Landscapes: Visualizing the Structure of Large Software Systems. In *Proc. of the IEEE TCVG Symposium on Visualization*, 261-266.

BORLAND TOGETHER, *www.borland.com/together*.

CHEN, K. and RAJICH, V. 2001. RIPPLES: tool for change in legacy software. In *Proc. of the IEEE Int'l Conf. on Software Maintenance*, 230-239.

EICK, S. C., STEFFEN, J. L., and SUMNER JR., E. E. 1992. Seesoft - A Tool for Visualizing Line Oriented Software Statistics. In *IEEE Trans. on Software Engineering*, 18, 11, 957-968.

ELLSON, J., GANSNER, E., KOUTSOFIOS, E., NORTH, and S., WOODHULL, G. 2001. Graphviz - Open Source Graph Drawing Tools. In *Proceedings of Int'l Symposium on Graph Drawing*, 483-484.

DEMEYER, S., DUCASSE, S., and LANZA, M. 1999. A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. In *IEEE Working Conf. on Reverse Engineering*, 175-186.

EISENBARTH, T., KOSCHKE, R., and SIMON, D. 2003. Locating Features in Source Code. In *IEEE Trans. on Software Engineering*, 29, 3, 210-224.

FEYNMAN, R. P., LEIGHTON, and R. B., SANDS, S. 2005. *The Feynman Lectures on Physics*. Addison-Wesley.

IBM RATIONAL SOFTWARE, *www.ibm.com/software/rational*.

IRWIN, W. and CHURCHER, N. 2003. Object oriented metrics: Precision tools and configurable visualisations. In *Proc. of the IEEE Symposium on Software Metrics*, 112-123.

JAVA PLATFORM DEBUGGER ARCHITECTURE, *java.sun.com*.

JONES, J. A., HARROLD, and M. J., STASKO, J. 2002. Visualization of Test Information to Assist Fault Localization. In *Proc. of the IEEE Int'l Conf. on Software Engineering*, 467-477.

LEWERENTZ, C. and SIMON, F. 2002. Metrics-Based 3D Visualization of Large Object-Oriented Programs. In *Proc. of the IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis*, 70-80.

LUKOIT, K., WILDE, N., STOWELL, S., and HENNESSEY, T. 2000. TraceGraph: Immediate Visual Location of Software Features. In *Proc. of the IEEE Int'l Conf. on Software Maintenance*, 33-39.

MARCUS, A., FENG, L., and MALETIC, J. I. 2003. Comprehension of Software Analysis Data Using 3D Visualization. In *Proc. of the IEEE Int'l Workshop on Program Comprehension*, 105-114.

MICHAUD, J., STOREY, M. A., and MÜLLER, H. 2001. Integrating Information Sources for Visualizing Java Programs. In *Proc. of the IEEE Int'l Conf. on Software Maintenance*, 250-259.

MOZILLA PROJECT, *www.mozilla.org*.

MÜLLER, H. A., TILLEY, S. R., and WONG, K. 1993. Understanding software systems using reverse engineering technology perspectives from the Rigi project. In *Proc. of the Centre for Advanced Studies on Collaborative research: software engineering*, 1, 217-226.

MURPHY, G. C., NOTKIN, D., and SULLIVAN, K. 1995. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *Proc. of the ACM Symposium on the Foundations of Software* Engineering, 18-28.

NETHERCOTE, N. and SEWARD, J. 2003. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science*, 89, 2, Elsevier Science Publishers.

NETHERCOTE, N. 2004. *Dynamic Binary Analysis and Instrumentation*. PhD. Thesis, University of Cambridge.

ORSO, A., JONES, J., and HARROLD, M. J. 2003. Visualization of Program-Execution Data for Deployed Software. In *Proc. of the ACM Symposium on Software Visualization*, 67-76.

OTTENSTEIN, K. J. and OTTENSTEIN, L. M. 1984. The program dependence graph in a software development environment. In *ACM SIGPLAN Notices*, 19, 5, 177-184.

PARKER, G., FRANCK, and G., WARE, C. 1998. Visualization of Large Nested Graphs in 3D: Navigation and Interaction. In *Journal of Visual Languages and Computing*, 9, 3, 299-317.

REISS, S. P. 2001. Bee/Hive: A Software Visualization Back End. In *Proc. of ICSE Workshop on Software Visualization*, 44-48.

WATERS, R. G. and CHIKOFSKY, E. 1994. Reverse engineering: progress along many dimensions. In *Communications of the ACM*, 37, 5, 22-25.

WEIDENDORFER, J., KOWARSCHIK, M., and TRINITIS, C. 2004. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *Int'l Conf. on Computational Science ICCS*.

WEISER, M. 1981. Program Slicing. In *Proc. of the IEEE Int'l Conf. on Software Engineering*.

WILDE, N., GOMEZ, J., GUST, T., and STRASBURG, D. 1992. Locating user functionality in old code. In *Int'l Conf. on Software Maintenance*, 200-205.

WILDE, N. and SCULLY, M. C. 1995. Software reconnaissance: mapping program features to code. In *Journal of Software Maintenance: Research and Practice*, 7, 1, 49-62.

XU, B., QIAN, J., ZHANG, X., WU, Z., and CHEN, L. 2005. A brief survey of program slicing. In *ACM SIGSOFT Software Engineering Notes*, 30, 2, 1-36.

ZHOU, C., SUMMERS, K. L., and CAUDELL, T. P. 2003. Graph visualization for the analysis of the structure and dynamics of extreme-scale supercomputers. In *Proc. of the ACM Symposium on Software visualization*, 143-149.